Programmation

Le langage Ada – 14 : tâches parallèles

14/07/2008

Posté par La rédaction | Signature : Yves Bailly
Tags : GLMF
0 Commentaire | Ajouter un commentaire

Retrouvez cet article dans : Linux Magazine 87

Après la course à la fréquence, il semble que la mode pour les ordinateurs personnels soit passée à la course au nombre de processeurs. Il devient dès lors très intéressant, pour les développeurs, de se pencher sur les possibilités offertes de mettre en œuvre un véritable parallélisme afin d'exploiter au mieux ces architectures. Comme dans bien d'autres domaines, le langage Ada était, dès sa conception, très en avance sur ce point par rapport aux autres langages étrangement plus populaires.

Nous allons voir dans cet article à quel point il est aisé de faire s'exécuter plusieurs portions de code en parallèle dans un programme Ada. Tandis que les langages plus répandus doivent avoir recours à des bibliothèques spécialisées, telles que pthread, boost-thread, etc., dès 1979 le langage Ada contient en lui-même tous les outils permettant l'exécution de tâches parallèles. Ainsi, le développeur n'a-t-il pas à se préoccuper de la mise en œuvre de ses tâches au niveau du système : ces aspects rebutants sont laissés à la charge du compilateur.

Un programme de fractales

Comme exemple de programme pouvant aisément être parallélisé, considérons le calcul d'une image fractale. Afin d'avoir un maximum de souplesse, ce programme sera décomposé en plusieurs paquetages, la formule de la fractale étant représentée par une interface abstraite. Voici la spécification du type Fractal :

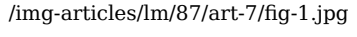
```
 1 with Ada.Numerics.Long_Complex_Types;
   2 use  Ada.Numerics.Long_Complex_Types;
   3 package Fractals is
   4    type Iterations is new Natural;
   5    type Fractal is interface;
   6    procedure Compute(fract   : in out Fractal;
   7                      from    : in     Complex;
   8                      bailout : in     Long_Float;
   9                      max_iter: in     Iterations;
  10                      nb_iters:    out Iterations)
  11       is abstract;
  12    function Is_Inside(fract  : in Fractal;
  13                       point  : in Complex;
  14                       bailout: in Long_Float)
  15       return Boolean
  16       is abstract;
  17 end Fractals;
```

Nous allons calculer des fractales " classiques ", c'est-à-dire dans l'ensemble des nombres complexes. La bibliothèque standard du langage Ada offre justement un type générique de nombres complexes, dans le paquetage Ada.Numerics.Generic_Complex_Types, le paramètre générique étant le type réel de base. La spécification précédente utilise une spécialisation de ce paquetage utilisant le type réel Long_Float (lignes 1 et 2), correspondant (en général) au type double du langage C.
Rappelons que le calcul d'une image fractale, au sens où nous l'entendons ici, consiste pour chaque pixel à évaluer une certaine formule de manière répétitive. Sont considérés dans l'ensemble fractale les pixels pour lesquels la formule n'aboutit pas à un éloignement à l'infini, c'est-à-dire ceux pour lesquels l'évaluation d'une certaine distance ne dépasse pas une certaine valeur (nommée " bailout "), après un certain nombre d'évaluations de la formule principale. Les autres pixels se voient attribués une couleur en fonction du nombre d'itérations nécessaires pour obtenir ce dépassement.
Aussi, commençons-nous par définir un type représentant les itérations, ligne 4, fondamentalement un entier positif ou nul. Ensuite, le type Fractal lui-même est déclaré, sous la forme d'une interface : chaque utilisateur pourra ainsi donner ses propres formules d'évaluation des pixels et des distances. La procédure Compute() effectue le calcul de la valeur fractale d'un point donné (paramètre from), soit jusqu'au dépassement de la distance de référence (paramètre bailout), soit pour un maximum de max_iters itérations de la formule.
Le nombre d'itérations effectuées est retourné dans le paramètre nb_iters. Enfin, la fonction Is_Inside() est chargée d'indiquer si un point se trouve dans l'ensemble fractale, c'est-à-dire d'évaluer la distance voulue et la comparer à la

valeur de référence ~~bailout~~. Ces deux sous-programmes sont déclarés abstraits (~~abstract~~), car ils sont des opérations primitives du type interface (donc abstrait) ~~Fractal~~.
Pour fixer les idées, définissons une dérivation du type Fractal permettant le calcul de la fractale classique de Mandelbrot, définie par la suite complexe suivante :

$$z_0 = c = \text{pixel}\ z_{n+1} = z_n^2 + c$$

Si la norme MathML vous pose des problèmes, voici cette même équation sous forme d'image :


/img-articles/lm/87/art-7/fig-1.jpg

Ce type ~~Mandelbrot~~ sera déclaré dans un paquetage ~~Mandelbrots~~ par :

```
6    type Mandelbrot is new Fractal with null record;
```

Naturellement, les sous-programmes ~~Compute()~~ et ~~Is_Inside()~~ devront être surdéfinis. Voici l'implémentation :

```
 1 package body Mandelbrots is
 2    procedure Compute(fract   : in out Mandelbrot;
 3                      from    : in     Complex;
 4                      bailout : in     Long_Float;
 5                      max_iter: in     Iterations;
 6                      nb_iters:    out Iterations) is
 7       iter_count: Iterations := 0;
 8       zn        : Complex := from;
 9    begin
10       while fract.Is_Inside(zn, bailout) and
11             (iter_count < max_iter)
12       loop
13          zn := zn*zn + from;
14          iter_count := iter_count + 1;
15       end loop;
16       nb_iters := iter_count;
17    end Compute;
18    function Is_Inside(fract  : in Mandelbrot;
19                       point  : in Complex;
20                       bailout: in Long_Float)
21       return Boolean is
22    begin
23       return (abs point) < bailout;
24    end Is_Inside;
25 end Mandelbrots;
```

L'image fractale elle-même sera représentée par un type ~~Fractal_Image~~, déclarée dans un paquetage éponyme :

```
 1 with Ada.Numerics.Long_Complex_Types;
 2 use  Ada.Numerics.Long_Complex_Types;
 3 with Fractals;
 4 use  Fractals;
 5 package Fractal_Images is
 6    type Fractal_Image(width : Positive;
 7                       height: Positive) is
 8       tagged private;
 9    procedure Set_Fractal(fi   : in out Fractal_Image;
10                          fract: access Fractal'Class);
11    procedure Set_Domain(fi    : in out Fractal_Image;
12                         center: in     Complex;
13                         radius: in     Long_Float);
14    procedure Set_Limits(fi       : in out Fractal_Image;
15                         bailout  : in     Long_Float;
16                         max_iters: in     Iterations);
17    procedure Compute_Pixel(fi: in out Fractal_Image;
18                            x : in     Positive;
19                            y : in     Positive);
20 private
21    type Iterations_Array is
22       array(Positive range <>,
23             Positive range <>) of Iterations;
24    type Fractal_Image(width : Positive;
25                       height: Positive) is
26    tagged record
27       fract    : access Fractal'Class;
28       min_z    : Complex;
29       max_z    : Complex;
30       step_z   : Complex;
31       bailout  : Long_Float := 4.0;
32       max_iters: Iterations := 720;
33       iters : Iterations_Array(1..width, 1..height);
34    end record;
35 end Fractal_Images;
```

Le type ~~Fractal_Image~~ est un type paramétré, les paramètres définissant la largeur et la hauteur de l'image (en pixels). Le domaine du plan complexe représenté par l'image est défini par un centre et un rayon, à la manière de l'excellent programme XaoS [2]. Les résultats des calculs pour chaque pixel, c'est-à-dire le nombre d'itérations effectuées de la formule fractale, sont stockés dans le tableau interne ~~iters~~ (ligne 33), de type ~~Iterations_Array~~ (lignes 21-23). Nous ne détaillerons pas l'implémentation, assez simple.
Enfin, voici le programme qui va effectivement calculer notre fractale :

```
 1 with Ada.Text_IO;
 2 use  Ada.Text_IO;
 3 with Ada.Numerics.Long_Complex_Types;
 4 use  Ada.Numerics.Long_Complex_Types;
 5 with Ada.Real_Time;
 6 use  Ada.Real_Time;
 7 with Fractal_Images;
 8 use  Fractal_Images;
 9 with Mandelbrots;
10 use  Mandelbrots;
11 procedure Fract is
12    fract_image: Fractal_Image(1024, 768);
13    mandel     : aliased Mandelbrot;
14    start      : Time;
15    stop       : Time;
16 begin
17    fract_image.Set_Fractal(mandel'Access);
18    fract_image.Set_Domain((-1.18764, -0.30278),
19                           0.00283869);
20    start := Clock;
21    for x in 1..fract_image.width
22    loop
23       for y in 1..fract_image.height
24       loop
25          fract_Image.Compute_Pixel(x, y);
26       end loop;
27    end loop;
28    stop := Clock;
29    Put_Line("Time:" &
30             Duration'Image(To_Duration(stop-start)));
31 end Fract;
```

Ce programme va donc calculer une portion de la fractale de Mandelbrot (ligne 17) dans une image de 1024x768 pixels (ligne 12), centrée au point de coordonnées (-1.18764, -0.30278) sur un rayon de 0.00283869 (lignes 18-19). Le temps de calcul est mesuré au moyen du type ~~Time~~ déclaré dans le paquetage standard ~~Ada.Real_Time~~ (lignes 5-6, 14-15, 20 et 28), puis affiché en secondes (lignes 29-30). Compilez ce programme au moyen de la simple commande :

```
 $ gnatmake fract
```

Si vous souhaitez appliquer certaines optimisations, faites-le de la même manière que pour gcc, par exemple :

```
$ gnatmake -O2 -march=pentium4 fract
```

Vous obtenez un exécutable nommé ~~fract~~. Voici quelques temps de calcul (en secondes), qui nous serviront de références pour la suite :

/img-articles/lm/87/art-7/t1.jpg

Les processeurs Celeron ne possèdent pratiquement rien pour faciliter l'exécution de tâches en parallèle, sans parler du cache interne de faible taille ; l'exécution de tâches parallèles ne devrait donc pas apporter beaucoup d'améliorations. La technologie HyperThreading [4] simule deux processeurs en un seul, offrant la possibilité de traiter beaucoup de données tout en réduisant l'impact des défauts de cache ; notre exemple ne traitant finalement que peu de données, on peut s'attendre à un gain limité en performance.
Enfin, le Pentium D est un processeur à double cœur, c'est-à-dire qu'en réalité deux processeurs sont contenus dans une même puce. Ce n'est donc plus de la simulation. Les temps de calculs devraient donc être sensiblement réduits en utilisant des tâches parallèles (un grand merci à Nicolas Boulay pour m'avoir permis d'abuser de son temps et de son matériel).

Fractale en parallèle

Passons donc aux choses sérieuses. Par nature, le calcul d'une image fractale telle que nous le faisons ici se prête bien au parallélisme : une première tâche peut s'occuper du calcul d'une première moitié de l'image, une seconde tâche de la

seconde moitié. Multipliez les tâches selon le nombre de processeurs dont vous disposez. Modifions notre programme fract ainsi (les premières clauses with et use sont inchangées) :

```
11 procedure Fract is
12    task type Fractal_Task is
13       entry Set_Interval(x_min: in Positive;
14                          x_max: in Positive);
15    end Fractal_Task;
16
17    fract_image: Fractal_Image(1024, 768);
```

Les lignes 12 à 15 déclarent un type tâche Fractal_Task, lequel contient un point d'entrée (entry) nommé Set_Interval(). La spécification d'un point d'entrée ressemble à celle d'une procédure, mais il ne s'agit pas d'un sous-programme.

Cela permet d'établir une communication synchrone avec une tâche en cours d'exécution, au moyen d'un rendez-vous. Ici, ce point d'entrée est destiné à définir un intervalle de l'image fractale à calculer par la tâche, d'où les deux paramètres x_min et x_max.

Remarquez que notre type est déclaré dans la procédure principale du programme : on pourrait également placer cette déclaration dans un paquetage dédié. Voici le code qui sera exécuté par les tâches de ce type :

```
19    task body Fractal_Task is
20       min_x: Positive;
21       max_x: Positive;
22    begin
23       accept Set_Interval(x_min: in Positive;
24                           x_max: in Positive)
25       do
26          min_x := x_min;
27          max_x := x_max;
28       end Set_Interval;
29       for x in min_x..max_x
30       loop
31          for y in 1..fract_image.height
32          loop
33             fract_image.Compute_Pixel(x, y);
34          end loop;
35       end loop;
36    end Fractal_Task;
```

Le corps du type tâche est introduit par task body, de façon similaire au corps d'un paquetage (ligne 19). Le code effectivement exécuté se trouve entre la paire begin/end des lignes 22 et 36. L'espace entre le task body et le begin peut contenir des déclarations de variables, de sous-programmes, etc. qui seront disponibles dans le code qui suit.

Nommons IT une instance de notre type tâche (nous verrons bientôt comment créer des instances de tâches). La première action de notre tâche est d'attendre que le point d'entrée Set_Interval() soit activé par une autre tâche, par la commande accept. Cela signifie que l'exécution de IT est bloquée à la ligne 23, jusqu'à ce que quelqu'un invoque le point d'entrée. Lorsque c'est le cas, l'appelant est lui-même bloqué durant l'exécution du code compris dans la paire do/end, lignes 25 à 28. Cela fait, IT et l'appelant poursuivent leur exécution normale. Dans le cas de IT, cela signifie calculer les valeurs d'itérations pour la zone spécifiée.

Remarquez que le corps de la tâche peut accéder à la variable fract_image, celle-ci ayant été déclarée auparavant, ligne 17. Si nous avions placé la déclaration de la tâche dans la spécification d'un paquetage, puis son corps dans le corps du paquetage, il aurait été nécessaire de communiquer une référence sur fract_image à la tâche.

Poursuivons notre programme :

```
38    mandel: aliased Mandelbrot;
39    start : Time;
40    stop  : Time;
41
42 begin
43    fract_image.Set_Fractal(mandel'Access);
44    fract_image.Set_Domain((-1.18764, -0.30278),
45                           0.00283869);
46    start := Clock;
47    declare
48       task_1: Fractal_Task;
49       task_2: Fractal_Task;
50    begin
51       task_1.Set_Interval(1,
52                           fract_image.width/2);
53       task_2.Set_Interval(fract_image.width/2+1,
54                           fract_image.width);
```

```
55       Put_Line("Computing...");
56    end;
57    stop := Clock;
58    Put_Line("Time:" &
59             Duration'Image(To_Duration(stop-start)));
60 end Fract;
```

Les variables globales sont inchangées, ainsi que l'initialisation de notre fractale. Le calcul est effectué dans le bloc compris entre les lignes 47 et 55.
Les lignes 48 et 49 déclarent tout simplement deux instances de notre type tâche ~~Fractal_Task~~, comme s'il s'agissait de variables classiques. Il est ainsi possible, par exemple, de déclarer un tableau de tâches. Ces deux tâches sont activées dès le début du corps du bloc, c'est-à-dire que l'exécution de leur corps commence dès le begin de la ligne 50. Cette exécution débutant par une instruction ~~accept~~, les deux tâches sont immédiatement bloquées. Leur exécution ne se poursuit qu'après l'invocation du point d'entrée ~~Set_Interval()~~, effectuée lignes 51 et 53. Le calcul simultané des deux moitiés de la fractale est alors entamé.
Si vous exécutez ce programme, vous constaterez que le message affiché ligne 55 apparaît presque immédiatement, puis que le programme semble bloqué avant l'affichage du temps de calcul. En fait, on peut voir notre programme principal comme une tâche principale, les deux tâches ~~task_1~~ et ~~task_2~~ étant des sous-tâches de celle-ci. Pour la tâche principale, l'appel à ~~Set_Interval()~~ ligne 51 est bloquant : elle ne poursuivra son exécution qu'après l'exécution de la petite portion de code du point d'entrée – qui est très rapide. L'appel ligne 53 est traité de la même façon. La suite du bloc consiste à afficher un message, puis à se terminer, ce qui implique la destruction des deux instances de notre type tâche.
L'attente se situe en fait dans cette terminaison : les instances des sous-tâches ne sont détruites à la sortie du bloc qu'une fois qu'elles ont terminé leur exécution, c'est-à-dire que l'exécution a atteint la fin de leur corps pour chacune. Le fait de déclarer ainsi les instances des tâches dans un bloc crée donc de fait une synchronisation entre la tâche principale et les deux sous-tâches.
Essayez de déclasser les déclarations de ~~task_1~~ et ~~task_2~~ avant la ligne 42 : vous verrez le premier message, puis le temps d'exécution s'afficher presque instantanément, puis le programme restera bloqué. L'attente aura en fait lieu sur la ligne 60, au moment où sont détruites les variables de la procédure principale, ainsi que les deux sous-tâches, qui ne sont pas encore terminées.
Pour fixer les idées, voici quelques temps de calcul de ce programme : (voir tableau 2)

/img-articles/lm/87/art-7/t2.jpg

Le Celeron donne un temps légèrement supérieur, ce qui n'est pas très étonnant : pour un système doté d'un unique processeur, l'exécution de tâches en parallèle peut s'avérer légèrement coûteuse du fait des changements de contextes. Toutefois, selon la charge du système et la taille des traitements à effectuer, deux tâches concurrentes peuvent s'avérer légèrement plus efficaces car le programme dans son ensemble peut se voir attribuer un peu plus de temps processeur. Par contre, l'HyperThreading du Pentium4 améliore le résultat d'environ 10%, bien qu'il ne s'agisse toujours que d'un unique processeur en simulant deux. Remarquez qu'un processeur double cœur, bien que légèrement moins rapide en fréquence pure, est plus efficace : la charge de calcul est mieux répartie, offrant un gain de plus de 20%.
Si vous vous demandez pourquoi le gain du double cœur n'avoisine pas les 50%, la raison en est que la zone choisie pour la fractale n'est pas symétrique : la moitié droite demande plus de calculs que la moitié gauche, ce que nous allons vérifier immédiatement (comme exercice simple, modifiez le code pour que la division soit horizontale et non plus verticale, puis constatez l'amélioration des performances).

Affichage : objets protégés

Calculer des fractales c'est bien, les voir c'est mieux. Nous allons pour cela utiliser la bibliothèque ~~Qt~~ version 4, au travers d'une interface nommée ~~" Qt4Ada "~~ Celle-ci, sur laquelle vous trouverez quelques détails supplémentaires plus loin, permet à un programme Ada d'accéder aux facilités de ~~Qt~~.
Notre nouveau programme va produire l'affichage suivant, une fois le calcul terminé :

/img-articles/lm/87/art-7/fig-2.jpg

À mesure de l'avancement des calculs, les pixels d'une image (de type ~~QPixmap~~) seront colorés selon le nombre d'itérations effectuées pour le point correspondant. À chaque fois qu'une colonne est terminée, l'affichage est rafraîchi, afin de visualiser la progression.
On pourrait donc imaginer simplement modifier notre programme ainsi :

```
    1 -- déclarations usuelles
...
   22 procedure Fract is
   23    task type Fractal_Task is
   24       entry Set_Interval(x_min: in Positive;
   25                          x_max: in Positive);
   26    end Fractal_Task;
   27    fract_image: Fractal_Image(1024, 768);
   28    pix        : QPixmap;
   29    label      : QLabel;
   30    black      : QColor;
   31    task body Fractal_Task is
   32       min_x  : Positive;
   33       max_x  : Positive;
   34       iter   : Iterations;
   35       color  : QColor;
   36       painter: QPainter;
   37       ok     : Boolean;
   38    begin
   39       painter.Setup;
   40       color.Setup;
   41       accept Set_Interval(x_min: in Positive;
   42                           x_max: in Positive)
   43       do
   44          min_x := x_min;
   45          max_x := x_max;
   46       end Set_Interval;
   47       for x in min_x..max_x
   48       loop
   49          for y in 1..fract_image.height
   50          loop
   51             fract_image.Compute_Pixel(x, y);
   52             iter := fract_image.Get_Pixel(x, y);
   53             if iter >= 720
   54             then
   55                color.Set_Hsv(0, 255, 0);
   56             else
   57                color.Set_Hsv(Integer(iter mod 360),
   58                              255,
   59                              255);
   60             end if;
   61             painter.Begin_Paint(pix, ok);
   62             painter.Set_Pen(color);
   63             painter.Draw_Point(x-1, y-1);
   64             painter.End_Paint(ok);
   65          end loop;
   66          label.Set_Pixmap(pix);
   67          label.Repaint;
   68       end loop;
   69    end Fractal_Task;
   70    mandel: aliased Mandelbrot;
   71    start : Time;
   72    stop  : Time;
   73    app   : aliased QApplication;
   74    ret   : Integer;
   75 begin
   76    app.Setup;
   77    black.Setup(0,0,0);
   78    pix.Setup(1024, 768);
   79    pix.Fill(black);
   80    label.Setup;
   81    label.Set_Pixmap(pix);
   82    label.Set_Fixed_Size(1024, 768);
   83    label.Show;
   84    Process_Events;
   85    -- la suite comme précédemment
...
  103    ret := Exec;
  104 end Fract;
```

Ligne 28 est déclarée une image de type ~~QPixmap~~, initialisée lignes 78-79. Le type ~~QLabel~~, dont une instance est déclarée ligne 29, permet d'afficher une telle image. Dans le corps de la tâche, chaque point de l'image est coloré selon le nombre d'itérations (lignes 51 à 64), l'affichage étant rafraîchi à la fin de chaque coordonnée x (lignes 66 et 67). Tout cela semble parfait.

Seulement, cela a toutes les chances de ne pas fonctionner. Du fait du multitâche, il n'existe aucune garantie quant à l'ordre dans lequel les instructions sont exécutées. En particulier, cela signifie que les deux tâches peuvent tenter de dessiner en même temps sur l'image, ou bien de l'afficher en même temps, ou pire encore, l'une peut tenter de l'afficher tandis que l'autre est en train de la modifier. Dans le meilleur des cas, vous n'aurez qu'une image partielle (avec des messages du type ~~X Error: BadGC~~ en provenance du serveur X), au pire, le programme plantera complètement.

Dans notre situation, l'affichage est une ressource critique qui supporte assez mal les accès simultanés. Une telle ressource doit être protégée, afin que les accès y soient dûment contrôlés et surtout sérialisés. Ce genre de principe se retrouve, par exemple, dans les systèmes de base de données ou les pilotes de matériels. Finalement, pour certaines opérations, on se retrouve à faire du mono-tâche.

On pourrait obtenir la protection recherchée en créant une nouvelle tâche destinée à l'affichage, qui prendrait la forme d'une boucle infinie dans laquelle la modification de l'image et le rafraîchissement de l'affichage seraient placés à la suite de points d'entrées. Mais cette façon de faire est lourde et laborieuse. Une bien meilleure solution est d'utiliser un objet protégé. Celui-ci est déclaré de la façon suivante:

```
27    protected type Pixmap is
28       procedure Setup;
29       entry Draw_Point(x   : in Integer;
30                        y   : in Integer;
31                        iter: in Iterations);
32       entry Update;
33    private
34       pix     : QPixmap;
35       label   : QLabel;
36       color   : QColor;
37       painter : QPainter;
38       ready   : Boolean := False;
39       drawing : Boolean := False;
40       updating: Boolean := False;
41    end Pixmap;
```

Un objet protégé, introduit par les mots-clefs ~~protected type~~, se compose d'une partie publique et d'une partie privée, comme un paquetage. La partie publique contient des fonctions, procédures et points d'entrées permettant d'accéder aux données de la partie privée. Ces trois types d'accès présentent les caractéristiques suivantes :

- les fonctions ne permettent qu'un accès en lecture seule aux données ; celles-ci n'étant pas modifiables au sein de la fonction, plusieurs tâches peuvent invoquer simultanément une fonction ;
- les procédures autorisent la modification des données ; afin de garantir l'intégrité de celles-ci, si plusieurs tâches invoquent une même procédure, elles sont placées dans une file d'attente et " servies " l'une après l'autre ;
- les points d'entrée (~~entry~~) sont similaires aux procédures, mais, en plus, elles ne peuvent être invoquées que lorsqu'une condition est vérifiée : elles sont gardées.

Les instances de ~~QLabel~~ et de ~~QPixmap~~ se trouvant dans la partie privée du type protégé ~~Pixmap~~ (lignes 34 et 35), elles ne sont pas accessibles depuis l'extérieur de ce type. Voyons l'implémentation (le corps) du type ~~Pixmap~~ :

```
43    protected body Pixmap is
44       procedure Setup is
45       begin
46          color.Setup(0,0,0);
47          pix.Setup(1024, 768);
48          pix.Fill(color);
49          label.Setup;
50          label.Set_Pixmap(pix);
51          label.Set_Fixed_Size(1024, 768);
52          painter.Setup;
53          label.Show;
54          ready := True;
55       end Setup;
```

La procédure ~~Setup()~~ ressemble à n'importe quelle autre procédure. Son rôle est d'initialiser l'image, l'affichage et les outils qui vont nous permettre de les modifier.

```
 56       entry Draw_Point(x   : in Integer;
57                         y   : in Integer;
58                         iter: in Iterations)
59          when ready and
60               (not drawing) and
61               (not updating) is
62          ok: Boolean;
63       begin
64          drawing := True;
65          if iter >= 720
66          then
67             color.Set_Hsv(0, 255, 0);
68          else
69             color.Set_Hsv(Integer(iter mod 360),
70                           255,
71                           255);
72          end if;
73          painter.Begin_Paint(pix, ok);
74          painter.Set_Pen(color);
75          painter.Draw_Point(x-1, y-1);
```

```
76          painter.End_Paint(ok);
77          drawing := False;
78       end Draw_Point;
```

Le point d'entrée ~~Draw_Point()~~ dessine dans l'image aux coordonnées données, dans une couleur dépendant du nombre d'itérations donné. Le gardien de ce point d'entrée figure lignes 59 à 61 : le code qui suit ne pourra être invoqué que si la variable ~~ready~~ vaut ~~True~~, et que la variable ~~drawing~~ vaut ~~False~~, et que la variable ~~updating~~ vaut ~~False~~. En fait, l'accès est un peu surprotégé, mais c'est pour illustrer. Durant l'exécution du code, les accès concurrents sont mis en attente : aucune autre tâche ne peut donc exécuter ce code en même temps.

```
79       entry Update
80          when ready and
81               (not drawing) and
82               (not updating) is
83       begin
84          updating := True;
85          label.Set_Pixmap(pix);
86          label.Repaint;
87          Process_Events;
88          updating := False;
89       end Update;
90    end Pixmap;
```

Le point d'entrée ~~Update()~~ fonctionne de la même manière. La protection des données est effectuée ainsi :

- si une tâche obtient l'accès à ~~Draw_Point()~~, aucune autre ne peut y accéder ; la variable drawing est alors placée à True : toute tâche demandant l'accès à ~~Update()~~ sera mise en attente, du fait du gardien (deuxième condition, ligne 81) ; ainsi on ne risque plus d'afficher l'image pendant qu'on la modifie ;
- de façon symétrique, lorsqu'une tâche obtient l'accès à ~~Update()~~, les accès à ~~Draw_Point()~~ seront mis en attente du fait du gardien ligne 61 : on ne risque plus de modifier l'image pendant qu'on l'affiche.

Cette technique des objets protégés permet une conception de haut niveau, le compilateur se chargeant d'ajouter le code nécessaire aux mises en attente et aux synchronisations. Il suffit alors de créer une instance de notre type protégé et de modifier légèrement le corps de notre tâche de calcul :

```
92    fract_image: Fractal_Image(1024, 768);
93
94    pix: Pixmap;
95
96    task body Fractal_Task is
97       min_x: Positive;
98       max_x: Positive;
99    begin
100      accept Set_Interval(x_min: in Positive;
101                          x_max: in Positive)
102      do
103         min_x := x_min;
104         max_x := x_max;
105      end Set_Interval;
106      for x in min_x..max_x
107      loop
108         for y in 1..fract_image.height
109         loop
110            fract_image.Compute_Pixel(x, y);
111            pix.Draw_Point(x,
112                           y,
113                           fract_image.Get_Pixel(x, y));
114         end loop;
115         pix.Update;
116      end loop;
117   end Fractal_Task;
```

Les appels à l'instance de l'objet protégé (déclarée ligne 94) apparaissent lignes 111-113 et 115. Le premier appel dessine un point, le second rafraîchit l'affichage. La procédure principale doit naturellement débuter par un appel à ~~pix.Setup~~, afin d'initialiser les données de l'objet protégé.

Voici ce que cela donne en cours de calcul :

/img-articles/lm/87/art-7/fig-3.jpg

Remarquez que les deux moitiés ne s'affichent pas à la même vitesse.
C'est parfaitement normal et même voulu : l'image n'est pas symétrique, la partie droite demande beaucoup plus de calculs que la partie gauche.
La seconde tâche progresse donc moins vite que la première, preuve indirecte que l'on a bien deux flots d'instructions

qui s'exécutent.
Voici les temps obtenus : (voir tableau 3)
Catastrophe, les durées explosent ! C'est normal, l'affichage répété d'une image aussi grande est extrêmement coûteux. Il serait donc préférable de limiter les rafraîchissements d'écran. Une solution consiste à utiliser...

/img-articles/lm/87/art-7/t3.jpg

Un chronomètre

L'idée est de ne rafraîchir l'affichage qu'à certains intervalles de temps, par exemple toutes les deux secondes. Pour cela, on va définir une nouvelle tâche chargée de cette action. Sa déclaration est toute simple :

```
  118    task type Timer is
  119       entry Start(tout: in Duration);
  120       entry Stop;
  121    end Timer;
```

Le point d'entrée ~~Start()~~ démarre le chronomètre, ~~Stop()~~ l'arrête. Le type prédéfini Duration représente une durée en secondes. Plus intéressante est l'implémentation de cette tâche :

```
  123    task body Timer is
  124       timeout: Duration;
  125       stopped: Boolean := False;
  126    begin
  127       accept Start(tout: in Duration)
  128       do
  129          timeout := tout;
  130       end Start;
  131       while not stopped
  132       loop
  133          select
  134             delay timeout;
  135             pix.Update;
  136          or
  137             accept Stop
  138             do
  139                stopped := True;
  140             end Stop;
  141          end select;
  142       end loop;
  143    end Timer;
```

L'exécution commence par l'attente du point d'entrée ~~Start()~~, avec en paramètre l'intervalle de temps à appliquer. Puis s'exécute une boucle jusqu'à ce que l'ordre d'arrêt soit donné, par le point d'entrée ~~Stop()~~.

Mais regardez le contenu de la boucle, lignes 133 à 141. Au sein d'une tâche, la structure introduite par select (ligne 133) permet d'effectuer un choix entre diverses portions de code, selon le premier événement qui survient – un peu comme une structure de choix multiple case. Ici, la ligne 135 ne sera exécutée qu'après l'expiration du délai demandé (l'attente ayant lieu ligne 134, par le mot-clef delay suivi d'une durée en secondes), à moins que durant cette attente le point d'entrée Stop() ne soit invoqué. En bouclant sur ce choix, on obtient bien un chronomètre.

À noter que l'attente ligne 134, bien qu'elle rende inactive la tâche, n'est pas bloquante : si ~~Stop()~~ est invoqué avant la fin de l'attente, alors le code correspondant reprend la main.

L'appel à ~~pix.Update~~ doit simplement être retiré du corps de la tâche de calcul. Le corps du programme principal doit enfin être légèrement adapté :

```
...
  149    ret   : Integer;
  150    chrono: Timer;
  151
  152 begin
  153    app.Setup;
...
  159    start := Clock;
  160    declare
  161       task_1: Fractal_Task;
  162       task_2: Fractal_Task;
  163    begin
  164       task_1.Set_Interval(1,
  165                           fract_image.width/2);
  166       task_2.Set_Interval(fract_image.width/2+1,
  167                           fract_image.width);
```

```
168       chrono.Start(2.0);
169       Put_Line(Standard_Error, "Computing...");
170    end;
171    chrono.Stop;
172    pix.Update;
173    stop := Clock;
```

...

Le chronomètre est déclaré ligne 150 et déclenché ligne 168. À l'issu du calcul, il est arrêté (ligne 171), un rafraîchissement de l'affichage étant nécessaire car il est possible (et même fort probable) que l'arrêt survienne alors que la fin de l'image n'a pas encore été affichée.
Voici ce que cela donne en termes de performances : (voir tableau 4, page suivante)
C'est mieux, une amélioration supplémentaire serait de ne dessiner dans l'image qu'aux moments choisis.

/img-articles/lm/87/art-7/t4.jpg

À propos de Qt4Ada

La bibliothèque ~~Qt4Ada~~ a été initiée par votre serviteur, pour les besoins de cet article et d'autres à venir.
Ce n'est encore qu'un projet au début de sa vie, mais au moment où ces lignes sont écrites les six premiers tutoriels de Qt4 ont été ré-implémentés en Ada avec succès. Vous trouverez Qt4Ada avec les codes sources de cet article [1].
Cette bibliothèque s'appuie, d'une part, sur la version 4.1.4 de Qt, d'autre part, sur le compilateur GNAT dans sa dernière mouture [3].
Pour compiler la bibliothèque, il est nécessaire de définir quelques variables d'environnement :

- ~~QTDIR~~ qui doit pointer vers l'endroit où vous avez installé ~~Qt4~~ ;
- ~~GNAT_HOME~~ qui doit pointer vers le répertoire dans lequel vous avez installé le compilateur GNAT ;
- ~~QT4ADA~~ qui doit pointer vers le répertoire contenant la bibliothèque ~~Qt4Ada~~ ;
- ~~PATH~~ doit impérativement commencer par ~~$GNAT_HOME/bin:$QTDIR/bin~~ ;
- ~~LD_LIBRARY_PATH~~ doit impérativement commencer par ~~$GNAT_HOME/lib:$QTDIR/lib:$QT4ADA/lib~~.

Vous pouvez utiliser le fichier ~~set_env.sh~~ à la racine de ~~Qt4Ada~~ pour vous faciliter la vie. Cela fait, un simple ~~make~~ devrait suffire à compiler la bibliothèque.
La compilation des deux derniers exemples de cet article, qui font appels justement à ~~Qt4Ada~~, se fait dans leur répertoire respectif par la commande ~~gnatmake -P fract.gpr~~.
Lorsqu'elle sera suffisamment avancée (à ce propos, toute aide est la bienvenue), cette bibliothèque fera très certainement l'objet d'un article spécifique.

Conclusion

On pourrait poursuivre longuement sur les tâches en Ada. Nous n'avons en réalité fait qu'effleurer la surface des possibilités offertes : points d'entrées conditionnels, contrôle de l'ordonnancement et des priorités entre tâches... Consultez le standard Ada pour plus de détails.
Le prochain article sera consacré aux structures de données génériques faisant désormais partie intégrante de la bibliothèque standard du langage Ada. Nous y retrouverons les traditionnels tableaux dynamiques et listes chaînées, mais également des tables de hachage et les itérateurs afférents. Si vous êtes un adepte de la bibliothèque générique standard du langage C++ (STL), vous vous retrouverez en pays connu, bien que peut-être plus attrayant.

Références :

- [1] Codes sources de l'article : http://kafka.fr.free.fr/articles/ada/sources_14.tar.bz2
- [2] XaoS : http://wmi.math.u-szeged.hu/~kovzol/xaos/doku.php
- [3] Le compilateur GNAT : https://libre2.adacore.com/
- [4] HyperThreading : http://fr.wikipedia.org/wiki/Hyperthreading

~~Retrouvez cet article dans : Linux Magazine 87~~